Acnet Setting Forwarding

Implementation notes

Mon, Mar 1, 2004

Most other front-ends must save Acnet device setting values to the Acnet database, because they have no local non-volatile memory in their hardware configuration. The Linac-IRM-PowerPC front-ends do have such nonvolatile memory in which setting values are saved. But the support for `DABBEL`-triggered downloading sometimes causes setting values from the Acnet database to get downloaded as well, even if the `DABBEL` change was only to modify a device and not create a new one. (This can occur when modifying the `SSDR`.) It has therefore been decided to forward settings to the Acnet database, even though the values will not normally be used, except for the `DABBEL`-related cases just described. This note describes some facets of the implementation in the system code.

The `ACReq` task is the path used for processing all Acnet protocol messages, each of which shares the same message queue on which `ACReq` waits. (Note that this ensures that a `SETDAT` message followed by a `RETDAT` message will always be executed in the original order.) The support for this new forwarding scheme needs another Acnet task name. The task name it targets is called `DBM...` (in `RAD-50` notation). So we choose to use the same task name, even though only reply messages from forwarding requests sent to the target `CDBS` node will be received, not request messages.

The protocol for these forwarded messages borrows heavily from the corresponding `SETDAT` message. A 3-word header is followed by an array of setting forward (`SF`) packets. The header includes a type code that is fixed at `0x0001`, the number of bytes expected in an ensuing reply message, and the number of `SF` packets in the array. Each `SF` packet includes a function code of `0x0300` followed by the corresponding `SETDAT` setting packet, but without the 8-byte `SSDN`. This means that the `SF` packet size is 10 bytes plus the size of the setting data.

The important need for forwarding settings to the Acnet database is for ordinary 2 or 4 byte setting values as well as for 20-byte alarm blocks. Accordingly, we limit this `SF` implementation to those `SETDAT` settings that include 20 bytes of data or less.

Setting forwarding logic flow

A `SETDAT` message can include a number of device setting packets, not all targeting the receiving node. A decision is made upon review of the target node numbers in all the setting packets on whether server-style logic is to be used, in which the entire `SETDAT` message is targeted to the single other node indicated for all the packets, or if more than one node is indicated, it is targeted to a multicast address to reach all the targeted nodes. The new forwarding logic applies only to `SETDAT` messages received that are *not* to be given server-style support.

In the loop that processes each local setting packet in a `SETDAT` message, a check is made for the setting data size being reasonable, as referred to above. If it is, then a `SF` packet is saved in a buffer. Then the setting action is performed, and if there are no errors detected, then the setting packet just built is entered into the `SF` message. Finally, after processing all of the device settings within the `SETDAT` message, the `SF` message header is set, and the accumulated `SF` message is queued to the network. When a reply ensues from `CDBS`, it is noted for diagnostic use, but no retry action is taken.

When one considers the larger picture, a `SETDAT` message sent to a front-end can elicit 3 messages from the front end. The first is the setting acknowledgment that is sent for any `SETDAT` message sent as a request (`REQ`) rather than a `USM`. The second is the new `SF` message described here. The third is an "accountability" record of the setting that is logged to Acnet so that operations can have a record of all settings made to front-ends to help assist in diagnosing strange behavior of accelerator devices.

The new implementation described here is to be installed in the system code, of course, so that all front-end nodes will include it. It is not optional, except that it is prompted only by `SETDAT` messages that include the `dbflag` set in the `SETDAT` header, so any node that does not receive an Acnet `SETDAT` message will also not generate the `SF` message. A front-end not installed in an Acnet installation will therefore not build `SF` messages.

More detail

The new logic is broken into several modules for ease in meshing it into the existing logic in the `ACReq` task. During `ACReq` task initialization, a new task name `DBM...` is opened for communication with `CDBS`. During nonserver processing of a `SETDAT` message, in the routine `SETNSERV`, for each device that is local, the routine `SFSAMPLE` is called to capture a copy of the current setting packet. (This allows for the possibility that the setting packet is modified during execution of the setting.) After successfully performing the setting for the current setting packet, the routine `SFENTER` is called, which places a `SF` packet into the `SF` message buffer. After all `SETDAT` setting packets have been processed, the routine `SFQUEUE` is called, which completes the `SF` message header and queues the `SF` message to the network. When a reply is received in response, the routine `SFREPLY` is called. All these routines are made to work together via a new set of local variables added to the `ACReq` task. Some of the new variables provide some internal diagnostics to assist during initial debugging as well as later on.

Post implementation

It turned out that the reply message actually has a 3-word header, consisting of a value of `0xAA55`, the number of `SF` packets, and the number of bytes of reply to be generated, not counting the 3-word header. The reply packets that follow are 2 words each, with the values `0x0210` and `0x0000`. The first is a "`DBM` pending" status, since it only means that the message was received; it has yet to be processed.

In order for the new feature to work, the front-end must be running the new system code, of course. But it is also necessary for each Acnet device-property, for which the forwarding feature is to be used, to have a flag set in the database. For alarm block properties, this flag is always set.

Knob adjustment problem

For knob adjustments, which often result in a series of `SETDAT` messages sent at 15 Hz, there is a problem, because updating the Acnet database at 15 Hz may place a strain on `CDBS`. Other front-ends are subject to the same problem, of course, but the `CAMAC` front-ends only forward settings if the `SETDAT` messages are sent as a `REQ` (thus prompting a reply) rather than a `USM`. The parameter page, the usual knob control client, sends a knob adjustment setting as a `USM`. Indeed, this serves to throttle the forwarding messages to `CDBS`, but it would seem inadvisable, as who is to say that the last setting made to a device was not made via a knob adjustment? Indeed, for some devices, knob adjustments may be the way they are most often set. (Think of tuning a Linac quad power supply according to the resulting effects on the beam.) One can easily end up with `CDBS` setting values that do not match the current actual setting values. It seems that another approach is needed.

Brian Hendricks suggested using a queue, emptied every few seconds, that holds `SF` packets that are intended to be sent to `CDBS`. If a new entry is to be placed into the queue, and there is already the same device represented therein, merely replace the setting value already in the buffer rather than add a new entry to the queue.

Logic for communicating `SF` messages to `CDBS` is already in place. When a new `SF` packet is appended to the buffer, let a counter be started to time out 5 seconds. If the buffer is full, call `NetQueue` to "get it out the door," and initialize a new empty buffer. In separate logic, sensitive to 15 Hz, count down the counter. When it reaches zero, empty the buffer by calling `NetQueue`. In this way, a flood of settings that fills the buffer will be sent out promptly. But if only one device is being set via knob control, the single entry is overwritten, until the time out transpires, and the buffer is shipped out.

One problem here is that time-sensitive logic cannot exist within the `ACReq` task, as it waits on a message queue, and many seconds may pass between executions of the task. One could send a special message through that queue to get `ACReq` to do the job of emptying the buffer.

The `ACReq` task waits on a message queue via a call to `NetCheck`. After a return from `NetCheck`, the message queue entry is examined. It can determine whether a special message is there and invoke `SFQUEUE`. But how does logic outside `ACReq` do this monitoring when it needs access to the `SF` variables within `ACReq`? We can simply post a pointer to the most recent `SF` message block in low memory. Everything necessary can be found within the block.

What about the buffer size that is allocated? Perhaps it should be a fixed size, rather than one that depends upon all the local settings within a single `SETDAT` message. A size of 512 bytes might be more than enough. Allowing 64 bytes for "infrastructure," this would allow for about 16 alarm blocks or about 40 ordinary 2-byte settings.

The periodic logic can be called from, say, the `Update` task, which always runs at 15 Hz. It needs access to the `SF` message block, to which `ACReq` can set a pointer in low memory. Within the message block, use a spare field to hold the timeout counter that will be initialized to zero when the block is created.

The message block is not always present; it is only created when `SFAPPEND` calls `SFALLOC`, which only happens when there is something to place in it. When `SFQUEUE` passes the block to the network, the low memory pointer to it should be set to `NULL`.

When a new entry is placed into the buffer (message block), the time out counter is initialized to 75, for example, which corresponds to 5 seconds at 15 Hz. If no more settings occur, the message block is queued to the network after 5 seconds. If repeated settings occur to the same device-property, as specified by the PIDI field, each merely updates the value in the buffer, but the timeout counter is not restarted. This means that the queuing to the network of the buffer will not be impeded just because someone continuously adjusts a device with a knob. The timeout counter should only be started when the first entry is placed into a new buffer. This ensures that any entry placed in the buffer will not wait longer than 5 seconds before it is delivered to CDBS. It also means that under continuous knob control settings, a SF message will only be sent every 5 seconds. The only time when it can be sent more often is when setting many devices at nearly the same time, in which the buffer goes out every time it is filled.

The periodic logic is simple. It checks the low memory variable for being non-NULL and therefore a pointer to a message block, in which case it decrements the counter field. If the count reaches zero, it sends a special message to the message queue on which the ACReq task awaits. To facilitate this, it can find this message queue id in another spare field in the message block structure. Note that although this simple code might reside in the ACReq module, it cannot access directly any of that task's static variables, since it runs as a part of the Update task. It only knows about the low memory pointer that is set and cleared by ACReq. Let this simple routine be called SFCYCLE.

When SFQUEUE sends the message block to the network, it clears the low memory pointer SETFBLK. Another block will only be created, and SETFBLK set to point to it, when a new SF packet needs to be entered into a message buffer. Within 5 seconds, that block will be sent to CDBS.

Replacement logic

Consider the replacement logic with some care. Scan the entries already in the buffer for a match on the PIDI (property index-device index). But it is possible that the length and/or offset do not match for the PIDI-matching entry. In that case, continue to scan for a later full match that matches both the PIDI and the length and offset values. Once a full match is found, replace the data. But if no full match is found, append the new partial matching entry.

Another way to say it is that a replacement can be used instead of an append only when there is a full match, and there is no further partial match later in the buffer. One could consider scanning from the end looking for a partial match. Then, if that partial match is not a full one, append must be used. But it is hard to scan through variable-size entries from the end; it is only reasonable to scan from the beginning. So, scan for a full match from the beginning. If one is found, continue the scan from that point for a later partial match. Only if no later partial match is found, perform a data replacement; otherwise, append the entry to the end.

Still there is a problem. Consider the following example of three settings. The first setting is made to a single field within a 20-byte alarm block. Then a second setting is made to the entire alarm block. Now a third setting is made to the same field that was set in the first setting. Clearly, the third setting value should not replace the first setting value, since the second setting would overwrite it. The point is that we must be very careful about re-ordering settings when it is possible to set parts of a structure. The solution here is to look for the last full match in the buffer, and only if that match is not followed by a partial match do we replace the data in the last full matching entry. If either no full match was found at all, or if the last full match is followed by a partial match, then we must append the new setting to the end of the buffer.

Modified plan of logic

At the start of SETNSERV, establish SETFDBF from the SETDAT message header. In SETNSERV, for each setting packet that is local, call SFSAMPLE to save a copy of it and set SETFDSZ to its data size if the dbflag in the SETDAT header is set and its size is suitable, else zero. Then execute the setting. If there is no error detected, and if SETFDSZ is nonzero, call SFENTER to add it to the buffer for reporting to CDBS.

In SFSAMPLE, if the dbflag in the SETDAT header is set, and if the current setting packet has 1–20 bytes of data, the current setting packet is copied into a sample buffer, which consists of SETFSAMP, SETFSAML, and SETSAMD. Then SETFDSZ is set to the size of the setting data, else 0.

In SFENTER, check the low memory pointer SETFBLK. If it is NULL, call SFALLOC to allocate a fixed-size SF message block. Now, if the pointer SETFBLK is non-NULL, scan through each SF packet already included, looking for the last packet matching both the PIDI and the length and offset. If one is not found, or if a further scan from that last matching entry finds a partial (PIDI-only) match, call SFAPPEND to add the new entry to the end. But if a full match was found, and no later partial match was found, replace the setting data in the last full matching entry with the setting data from the new packet.

In SFAPPEND, check that there is room in the fixed size buffer for the new entry. If there is not room, call SFQUEUE to queue the message block to the network, then call SFALLOC to create a new empty buffer. Now, if there is room for the new entry to fit, do the actual append step to copy from the sampled buffer, advance the message block MSGLEN field accordingly, and increment SETFWDN.

In SFQUEUE, check the low memory SETFBLK to be sure a message block exists, then call NetQueue to queue it for network transmission. Clear SETFBLK, SETFWDP, and SETFWDN.

In SAALLOC, allocate a fixed size message block of, say, 512 bytes, and store its address in SETFBLK and SETFWDP and clear SETFWDN. Initialize the message block, including setting the SETFTOC timeout counter field to 75, say, to provide for a 5 second timeout. Copy ACRQXID to the SETFQID field., to be used by SFCYCLE to wake up the ACReq task.

In SFCYCLE, which is invoked by the Update task every 15 Hz cycle, check that SETFBLK is non-NULL, and if it is, check the SETFTOC timeout counter field. If the counter is zero, send a special message to the message queue by using the SETFQID field; otherwise, merely decrement the counter.

In ACReq, calling NetCheck, call SFQUEUE if the special message is found.